Formal Semantics, Compilation and Execution of the GALS Programming Language DSystemJ

Avinash Malik, Alain Girault, Zoran Salcic

Formal Semantics, Compilation and Execution of the GALS Programming Language DSystemJ

Avinash Malik, Alain Girault, and Zoran Salcic, *Senior Member*, *IEEE*

Abstract—The paper presents a programming language, DSystemJ, for dynamic distributed *Globally Asynchronous Locally Synchronous* (GALS) systems, its formal model of computation, formal syntax and semantics, its compilation and implementation. The language is aimed at dynamic distributed systems, which use socket based communication protocols for communicating between components. DSystemJ allows the creation and control at runtime of asynchronous processes called clock-domains, their mobility on a distributed execution platform, as well as the runtime reconfiguration of the system's functionality and topology. As DSystemJ is based on a GALS model of computation and has a formal semantics, it offers very safe mechanisms for implementation of distributed systems, as well as potential for their formal verification. The details and principles of its compilation, as well as its required runtime support are described. The runtime support is implemented in the SystemJ GALS language that can be considered as a static subset of DSystemJ.

Index Terms—GALS systems, distributed programming, dynamic reconfiguration, weak mobility, formal model of computation, semantics, CSP, π-calculus, SystemJ, DSystemJ.

1 Introduction

An increasing number of computing applications are concurrent in nature and the only way to describe them efficiently is to use new concurrent programming languages that allow explicit use of concurrency. Sometimes, concurrency is a natural way to describe system operation, while in other cases, it is natural due to the nature of the execution platform such as distributed networked systems. Some typical examples of such applications are sensor networks capable of dealing with nodes being attached or detached *at runtime*, and ad hoc collaborative systems in which participants *dynamically* enter and exit from a joint activity, such as multiplayer gaming, or document editing environments.

We define a superset of distributed systems that we target, called *dynamic distributed systems* (DDS), capable of creating, terminating, migrating, and managing processes at runtime in a distributed environment. Existing programming languages have not kept pace with this increase in demand of concurrent applications. In fact, concurrent programming with standard languages is still considered difficult [1]. The main reason for this difficulty arises from the fact that the concurrent programming advocated by standard languages requires programmers to deal with synchronization and communication between concurrent processes at a very low level of abstraction, thus diverting them from the actual system design.

Over the years, a number of programming techniques and languages have been proposed to make concurrent and especially distributed programming more productive. The *de facto* standard for distributed computing is the *Message Passing Interface* (MPI) specification [2]. MPI is usually implemented as a library providing an *Application Programming Interface*, which can be used from different languages. Other approaches include mobile agent systems, like JADE [3] based on Java, which are specifically designed to take advantage of Java's portability. Yet, these runtime libraries are often very heavy in terms of memory footprint and resource requirements, thereby making them unsuitable for systems with less powerful computing nodes, like those used in sensor networks.

The above mentioned approaches model systems with asynchronous concurrency without being based on a formal *Model of Computation* (MoC) and a formal semantics, or provide only a partial formal semantics. We believe that formal semantics is essential for faithful compilation and reasoning about the program. Formal semantics is the cornerstone for state space exploration techniques [4], which can be used for formal verification—an important step in building trustworthy, highly reliable systems. More generally, we advocate that formal semantics is essential in languages that are used to describe DDSs, as it offers the potential to reason about the correctness of such complex systems.

A number of formal languages, equipped with a formal semantics, have been introduced, like *Communicating Sequential Processes* [5] (e.g., Occam [6]), π-calculus [7]

- *A. Malik is with the IBM Research Dublin and the Department of Computer Science and Statistics, Trinity College Dublin, Ireland. E-mail: avimalik@ie.ibm.com.*
- *A. Girault is with the INRIA Grenoble Rhône-Alpes and LIG laboratory, POP ART team, Grenoble, France. E-mail: alain.girault@inria.fr.*
- *Z. Salcic is with the University of Auckland, New Zealand. E-mail: z.salcic@auckland.ac.nz.*

Manuscript received 15 Dec. 2010; revised 14 Sept. 2011; accepted 19 Sept. 2011; published online 19 Oct. 2011.
Recommended for acceptance by F. Mueller.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2010-12-0720.
Digital Object Identifier no. 10.1109/TPDS.2011.258.

(e.g., Occam-π [8]), Join-calculus [9] (e.g., JoCaml [10]), and Actor Models [11] (e.g., ActorFoundry [12]). All these approaches have some merits and some disadvantages. Occam and Occam-π are able to model static and dynamic distributed systems, respectively, but they are unable to express complex data transformations and cannot abstractly express data fusion from multiple sources.

JoCaml is able to model DDSs with complex data transformations, thanks to the ML programming language as its base. However, it is unable to express data fusion abstractly. Also, unlike Occam-π it does not allow mobility of processes at runtime.

Finally, actor-based languages and libraries, like ActorFoundry, Scala [13], and Erlang [14], are not designed to execute efficiently on distributed systems or lack a number of the above mentioned capabilities.

In this paper, we introduce a new programming language aimed at DDSs communicating via socket-based networks, called DSystemJ. It is a conservative extension of the *Globally Asynchronous Locally Synchronous* (GALS) language SystemJ [15]. DSystemJ extends SystemJ with new features to deal with the dynamics of asynchronous processes, called clock-domains. Like SystemJ, it also allows each clock-domain (CD) to be expressed as a composition of multiple synchronous concurrent processes, called reactions. Thanks to its GALS MoC, DSystemJ is able to model a larger class of systems than any of the above-mentioned approaches. DSystemJ allows the designer to **fork** a new clock-domain at runtime (dynamic creation), it provides convenient means to describe **weak** CD **mobility**, it provides an abstract means to describe data fusion, reactive programs, and complex control situations (**synchrony/asynchrony**), while at the same time mixing them with complex data computations in standard Java (**heterogeneity**). Finally, DSystemJ's implementation of communication between reactions in different CDs is based on *Communicating Sequential Processes*. DSystemJ is designed to work in a fully distributed memory environment, without a single entity having the complete knowledge of the system and its state. This property, along with self contained CD execution, makes systems implemented using DSystemJ adhere to the principle of no single point of failure, i.e., disruptions in the execution of a single CD do not prevent the rest of the program from continuing its execution, thereby making the system resilient to failures. Furthermore, DSystemJ also allows reinstantiation of failed CDs, thereby providing fault recovery capabilities.

After the presentation of the language itself, we focus on the key aspects of language, compilation, and implementation, including the description of the runtime environment necessary to support the dynamic nature of the language.

The rest of the paper is organized as follows: Section 2 presents the DSystemJ MoC, syntax, and intuitive semantics. Section 3 gives an example of a DSystemJ program, highlighting the main features of the language. The formal semantics and MoC are presented next in Section 4. Section 5 explains the compilation procedure. Section 6 provides an overview of the DSystemJ runtime system and associated libraries. A detailed comparison between DSystemJ and currently available languages and libraries is provided in Section 7. Section 8 gives a quantitative comparison. Finally, we end the paper in Section 9 with the conclusions and future work directions.

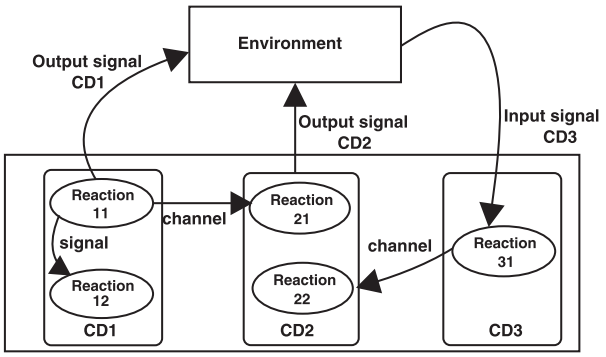


Fig. 1. A general representation of a SystemJ example.

2 DSystemJ: Model of Computation, Syntax and Intuitive Semantics

DSystemJ is a conservative extension of SystemJ [15] and hence it follows the GALS MoC of the SystemJ language.

A SystemJ program consists of a set of clock-domains, (in Fig. 1, CD1, CD2, and CD3 are CDs) composed using the asynchronous parallel operator ($><$) and executing at unrelated logical clock ticks (from here on referred to as tick). CDs synchronize and communicate with each other using **point to point** channels with CSP [5] style rendezvous for synchronization and data transfer. Each CD itself consists of one or more processes, called reactions, executing in lockstep, i.e., at the CD's tick. In Fig. 1, Reaction 11, Reaction 12, etc., are synchronous parallel reactions. The reactions are combined and controlled using the synchronous parallel composition operator ($\|$). Reactions within the same CD communicate using the synchronous **broadcast** mechanism over objects called signals. Finally, a SystemJ program interacts with its environment through a set of input and output signals. The synchronous statements, reactions, and operations on signals, and asynchronous statements, CDs, and channels, are together responsible for the control-flow of SystemJ programs. The data driven computations are written in Java and considered instantaneous in terms of advancement of logical clock (tick consumption).

A DSystemJ program extends this with the ability to fork new CDs at runtime (dynamic process creation) and pass CDs over channels (weak process mobility).

2.1 DSystemJ Syntax

DSystemJ combines features from Esterel [16], CSP [5], and π-calculus [7] with the Java programming language. Tables 1 and 2 show the SystemJ and DSystemJ kernel statements and their meanings. A more detailed explanation of these constructs is presented later in the sections.

Signals are the most basic means of communication in a DSystemJ program; they have a status and possibly a value. Signals can be either local or interface signals, interface signals are qualified with either the `input` or the `output` keywords and are used for communication with the

TABLE 1
The SystemJ Kernel Statements and Their Meaning

Kernel Statements	Meaning
[**input**] [**output**] [**type**] **signal** S	declare signal S
emit S [(value)]	broadcast signal S
present (S) {p} else {q}	do p if S is present, else do q
abort (S) {p}	preempt program p if S is present
suspend (S){p}	suspend p if S is present
trap (T){p…**exit** T…}	preempt p if exit is executed
p||q	run p and q in lock-step
p><q	run p and q asynchronously
send C([value])	send a value through C, blocking send
receive C()	receive a value through C, blocking receive
pause	finish a tick and communicate with environment

TABLE 2
The Kernel Statements in DSystemJ and Their Meaning

Kernel Statements	Meaning
unique-name → CD([args])	declare a closure with the named clock-domain CD
unique-name → {}	declare a closure with the unnamed (anonymous) clock-domain
run unique-name([args])	run CD unique-name
run #channel-name([args])	run CD received via channel channel-name
[**input**] [**output**] [**type**] **channel** C	declare channel C

environment, while local signals are used for communication between concurrent reactions within a single CD (see Fig. 1). A signal emission broadcasts the signal throughout its CD, making it instantaneously visible to all the reactions running in lock-step within that CD. The emission of an **output** signal makes it visible to the environment, too (see Fig. 1). A signal emission can be pure or include a value, which can be of any Java data type. The **present** instruction is used to check the presence of a signal, while **abort** and **suspend** instructions are used for preemption. The **trap** and **exit** statements together implement user defined preemptions, as opposed to environment based ones through signals (**abort**, **suspend**).

There are multiple ways to represent concurrency in DSystemJ. The $\|$ and $><$ operators initialize synchronous parallel reactions and CDs, respectively, at program startup. The **run** statement allows designers to instantiate new CDs at runtime. The channels are used to communicate between reactions in different CDs (see Fig. 1). The **send** and **receive** statements together implement a rendezvous (blocking sending and receiving) style communication through channels. Finally, the **pause** statement marks the completion of a tick of a single CD or a reaction. At the start of the tick of each CD, the CD's input signals are sampled from the environment by the program; next the required transitions are computed, and, finally, the CD's output signals are emitted to the environment at the end of the CD tick, thereby implementing a state machine. All the syntactic constructs presented in Tables 1 and 2 can be freely intermixed with most of the Java constructs (refer [15] for complete description).

2.2 Intuitive Semantics of Kernel Statements

In this section, we describe the intuitive semantics of the kernel constructs introduced in the DSystemJ language (Table 2). The reader is referred to [15] for a detailed explanation about the rest of the kernel statements (Table 1).

2.2.1 The Unique-Name → CD([args]) and Unique-Name → {} Constructs

DSystemJ like SystemJ allows a programmer to declare: 1) named CDs, which can be reused by invoking them at different instances (like named functions in functional programming languages), or 2) unnamed or anonymous CDs, which cannot be invoked at different instances, and are equivalent to lambda functions in functional programming languages. The → construct builds a closure of a named or anonymous CD. The → delimited name ("unique-name") can then be used to fork or send the CD code via channels. These unique-names have a *global scope* in the system, i.e., they are visible to all the reactions and other CDs, including themselves. Every closure keeps a separate copy of the enclosed variables, which can also include reactive constructs like signals. The enclosed variables are not shared among the closures. The enclosed variables are replaced with new values when forking CDs via the **run** statement. Finally, the → operator can be applied several times with the same named CD but with different arguments.

2.2.2 The `run` Unique-Name ([args]) and `run` #Channel-Name([args]) Constructs

The **run** construct is used to dynamically fork CDs. The version "**run** unique-name ([args])" forks the CD already registered with the runtime system, while the version "**run** #channel-name ([args])" forks a CD received via the channel "channel-name." The **run** statement performs a *rendezvous* with the runtime system, asking it to fork the required CD. The **run** statement takes a finite number of ticks to succeed, but the number of ticks cannot be statically determined. The "tick" here refers to the logical tick of the CD that invokes the **run** statement. Any CD forked via the **run** statement starts from its initial state, i.e., DSystemJ does not allow one to save the current state of a forked CD and hence, only weak mobility is possible (mobility of code, but not the state of the program).

2.2.3 The `send` C(Unique-Name) and `receive` C Constructs

The **send** statement in DSystemJ is similar to the **send** statement in SystemJ. It performs a rendezvous with the **receive** statement on the same channel-name. Both are blocking. In SystemJ, the **send** and **receive** statements can pass any Java object. DSystemJ provides the syntactic sugar of being able to pass the CD's unique-name itself to implement weak CD mobility, rather than manually constructing a Java object containing the marshaled CD code.

The major difference between DSystemJ and SystemJ rendezvous communication stems from the fact that DSystemJ rendezvous communication is not restricted to being point-to-point (linear). Indeed, DSystemJ allows one to many (single sender-multiple receivers), many to one (multiple senders-single receiver), and many to many (multiple senders and receivers) rendezvous between multiple participants. Listing 1 further illustrates this point.

Listing 1. An example of non linear channel communication in DSystemJ

```
1  //Example of Many to One
2  //non linear channel communication on channel "M".
3  //Recall that >< is the asynchronous operator
4
5
6  //CD P running on machine 1 sends itself via channel C
7  //In parallel it also sends values via channel M
8  P → send C(P); || while(true) send M(4);}}
9  ><
10 //CD Q running on machine 2 gets the value via channel M
11 Q → while(true) receive M;}
12 ><
13 // CD R running on machine 3
14 //forks CD P obtained via channel C
15 //and finishes itself. But, now CD P
16 //runs on machine-3 as well, blocking
17 //on channel C due to lack of a receiver and also sending
   values on
18 //channel M
19 R → receive C; run #C();}
```

In the multiparticipant case, the DSystemJ runtime *nondeterministically* chooses a partner to rendezvous with, similar to the **`select`** statement in ADA [17]. The nondeterministic selection of a rendezvous partner in a multiparticipant scenario raises fairness issues. Indeed, multiparticipant rendezvous can introduce starvation in a system. For example, in Listing 1 above, the **`receive`** statement (line 11) might always choose to rendezvous with the **`send`** on machine 1, thereby starving the CD on machine 3.

The DSystemJ compiler is able to statically detect only some starvation situations in multiparticipant rendezvous [18]. As DSystemJ is targeted toward DDSs with the goal of *no single point of failure*, there is no single entity in the system having the complete knowledge of the system at runtime. As a result, DSystemJ does not guarantee process fairness. Instead, the developer is advised to use separate channels by creating them at runtime thanks to the **`channel`** construct.

DSystemJ's fault tolerance and recovery capabilities are also highlighted in the example of Listing 1. If any of the CDs (say `P`) fails during program execution, other CDs being completely self contained keep on progressing with their individual execution. In this particular case, `Q` and `R` both keep on waiting for a rendezvous with `P`, which never occurs due to failure of `P` . `P` can be reinstantiated while the other CDs are running; upon it's reinstantiation, `P` communicates with the running CDs and the program proceeds further. Such fault recovery capabilities are essential for implementing robust DDSs.

3 Language Features and Example

In this section, we present a security surveillance system designed with the DSystemJ programming language. This example highlights: 1) the motivation behind the design of DSystemJ language constructs for programming complex systems, 2) the reduction in development and debugging time, and finally, 3) familiarization with DSystemJ syntactic constructs.

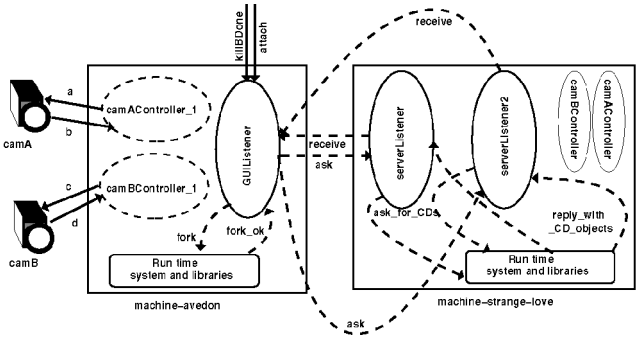


Fig. 2. Pictorial representation of a security surveillance system. The thick solid ellipses are CDs initialized at the start of the system. The thin solid ellipses are CDs present in the system as code but not yet running. The dashed ellipses are instantiations of the CD code forked at runtime.

Fig. 2 shows a security surveillance system. This system consists of two Internet enabled PTZ (Pan/Tilt/Zoom) cameras (`camA` and `camB`), capable of following moving objects in an indoor environment. There are two physical machines; `avedon`, which is connected to the cameras and acts as their controller, and `strange-love`, which acts as the code repository holding the control code for the cameras. There can be a number of other machines acting as video servers, authentication servers, etc. For simplicity we have shown only two physical machines and cameras, but the system can consist of any number of cameras and machines. There is also a Graphical User Interface (GUI) with which the user interacts with the cameras observing the object being tracked and also, giving commands as needed and so on. This GUI communicates with the cameras via `avedon`. These two machines communicate with each other, with the cameras and with the user to carry out surveillance. The whole system is designed/implemented with the DSystemJ language.

Our system is capable of:

1. *Attaching a camera*: Either of the cameras can be attached after the system is up and running. If `camA` is attached at runtime, `avedon` detects this event and asks `strange-love` to provide the controller code for `camA`, via channels `askA` and `receiveA` as shown in Listing 2 lines 18 and 21, and Listing 3 lines 16-20 and 27-30. CD forking in DSystemJ is as easy as calling the **`run`** statement. This also highlights the weak process mobility primitives built into the language; sending and receiving CDs via channels. Such abstraction and design comfort is impossible to achieve with the SystemJ language alone. SystemJ does not provide any means of forking CDs at runtime, so, a designer would need to implement process mobility explicitly using Java's code marshaling abilities. This in turn would increase the number of lines of code (LOC), resulting in hard to debug and maintain code.
2. *Detaching a camera at runtime*: Our system is able to detect and respond to the detachment of cameras at runtime, without affecting the rest of the system. This ability stems from the fact that all camera

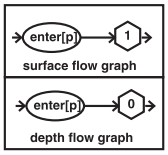


Fig. 3. The **pause** statement.

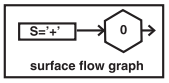


Fig. 4. The **emit** statement.

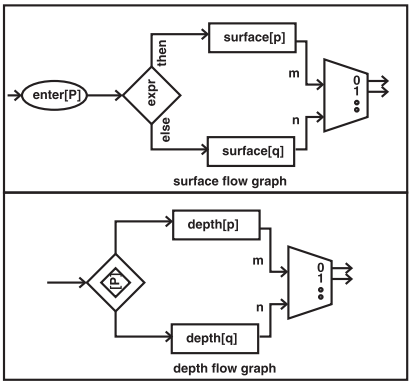


Fig. 5. The **present** statement.

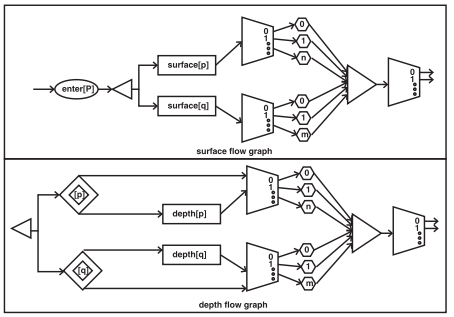


Fig. 6. The synchronous parallel operator.

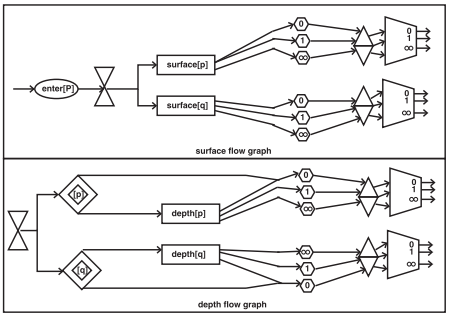


Fig. 7. The asynchronous parallel operator.

controllers are modeled as CDs, which run independently of each other. DSystemJ inherits this capability from the SystemJ language.

3. *Efficiently utilizing distributed and multicore systems*: DSystemJ CDs are compiled into Java threads (for a multicore system) and separate Java processes (for a distributed system). These asynchronous processes/threads can then make use of the hardware level parallelism so prevalent in recent times. Furthermore, a designer does not have to deal with the low-level details such as synchronization and locking. Instead, DSystemJ's **send**/**receive** based communication is simpler to comprehend. DSystemJ's multiparticipant rendezvous is utilized in our security surveillance example. In this example, two `serverListener` CDs are instantiated on machine `strange-love` (Listing 2, lines 16-31). When the `GUIListener` CD asks for a specific camera controller CD, either of the `serverListener` might respond to this request. The multiple serverListener CDs have the following roles: a) Provide fault tolerance mechanism: if either of the `serverListener` CD fails, the other can respond to the request from `GUIListener`, thereby allowing the program to proceed further without complete system halt. b) In the general case, multiple `serverListener` CDs run on different machines, the request from `GUIListener` CD is responded to by the closest `serverListener` CD, or with the fastest communication link to the requesting `GUIListener` CD, thereby increasing the throughput of the system. c) Finally, multiple `serverListener` CDs would be essential in a large distributed system for load balancing requests from multiple GUIListener CDs. Finally, SystemJ also provides the asynchronous >< construct to create CDs; this asynchronous parallelism is only logical, i.e., all the parallelism is compiled into a single threaded code [15], and hence, SystemJ is unable to take advantage of the multicore and distributed architecture.
4. *Abstraction of system design and implementation*: DSystemJ's communication mechanism based on signals essentially decouples system design from low level implementation. Thus, even if the implementation changes, the design remains the same. This flexibility can be seen from communication mechanisms shown in Listing 3 lines 7, 13, 25, and 38-40.

Listing 2. A dynamic security surveillance system

```
1  system{
2    interface{
3    //The signals and channels that are used for
4    //communication with the environment and between the
     various CDs.
5    input Object channel askA,askB,receiveA,receiveB;
6    output Object channel askA,askB,receiveA,receiveB;
7    input String signal attach,ctrlA,ctrlB,killBDone,killB;
8    output String signal attachMessage,ctrlMessageA,
     ctrlMessageB;
9  }
10 {
11 //The ruby GUI listener CD
12   GUIListener→ GUIListener(askA,askB,receiveA,
     receiveB,attach,
13   killBDone)
14   ><
15   //CD server listener
16   serverListener →{
17     {
18     while(true){receive askA; send receiveA
       (camAController); }
19     }||
```

```
20    {
21    while(true){receive askB; send
      receiveB(camBController);}
22    }}
23    ><
24  //CD server listener2
25  serverListener2→ {
26    {
27    while(true){receive askA; send
      receiveA(camAController);}
28    }||
29    {
30    while(true){receive askB; send
      receiveB(camBController);}
31    }}
32  ><
33  //The camera A controller
34  camAController→ {//Alternatively move camera A left
    and right
35 ><
36 //The camera B controller
37 camBController→ {//Alternatively move camera B
   right and left}
38 }
39 }
```

Listing 3. The GUI listener CD

```
1  reaction GUIListener(output Object channel askA,
   output Object
2   channel askB, input Object channel receiveA, input
    Object channel
3   receiveB, input String signal attach, input String
    signal killBDone){
4   boolean aDone=false,bDone=false;
5   {
6   while(true){
7    await(attach);
8    String name = ((String)#attach);
9    //Asked to attach controller for camera A?
10   if(name.equals("ATTACH_A")){
11    //Is camera A controller already attached in runtime?
12    if(Helper.exists("camControl.camAController")
      && aDone)
13     emit attachMessage("A exists");
14    //Get the camera A controller from the server.
15    else{
16     send askA("camControl.camAController");
      //The fully qualified name
17     pause;
18     receive receiveA; //Received the camera controller
      A CD
19     run #receiveA();
20     aDone=true;
21     emit attachMessage("A is now controllable");}}
22    //Same as for A
23    else if(name.equals("ATTACH_B")){
24     if(Helper.exists("camControl.camBController")
       && bDone)
25      emit attachMessage("B exists");
26     else{
27      send askB("camControl.camAController");
28      pause;
29      receive receiveB;
30      run #receiveB();
31      bDone=true;
32      emit attachMessage("B is now controllable");
33    }}}
34  }||
35  {
36  while(true){
37      //Get the Kill B signal from GUI
38      await(killBDone);
39      bDone=false; //Set the B killed boolean to false
40      emit attachMessage("B killed");
41      pause;}}
42  }
```

4 Formal Semantics

This section presents the formal semantics and the MoC of DSystemJ. Both are described in terms of SystemJ MoC and microstep semantics, which we describe first in Section 4.1. These microstep semantics can be used to construct the macrostep semantics of compiled DSystemJ programs. The macrostep operational semantics are essential for formal reasoning and *Worst Case Reaction Time* (WCRT) analysis of DSystemJ programs.

4.1 Semantics of SystemJ

All of SystemJ's constructs utilize a structural translation scheme. We use one or more semantical rule(s) to rewrite the reactive control and Java data statements. Such a translation scheme helps us obtain a direct intermediate representation of the program from which back-end code can be efficiently generated. The semantical rewrite rules presented are very fine grained, being targeted toward compiler construction, and thus, we also call them microstep kernel semantics.

Let p be a SystemJ kernel statement, we write

$$term(p), data \xrightarrow[E,E_c]{k,e} term'(p), data', \tag{1}$$

where $term(p)$ and $term'(p)$ represent the antecedent and consequent states of p, respectively, during a microstep transition. Term e represents the signals that are emitted during the transition, and if none are emitted then it takes the value $\perp$. Term $data$ represent the value stores attached to the statement p before transition and $data'$ after the transition. Term k represents the termination code. It has a value of $\perp$, i.e., unknown, if p does not generate a termination code after this transition, else, an integer value within $[0, \infty]$. A termination code of 0 represents the completion of the reaction, 1 represents the completion of a "local" tick, a termination code in the interval $[2, \infty)$ is reserved for preemptions based on **`trap/exit`** constructs, and finally a termination code of ∞ shows an unresolved signal dependency, e.g., in case of **`emit`** and **`present`** statements.

The signal sensitivity set E is the status of all the signals used in p, but declared somewhere else. The channel sensitivity set E_c is the status of all the channel ports used in p, but declared somewhere else. $term(p)$ takes a transition depending upon the status of these sensitivity sets. For a number n of channels, there are $2 \times n$ input and output ports, corresponding to the receiving and sending ends. For a channel C, $E_c = \{\{Cw_s, Cw_r, Cp_s\}, \{Cr_s, Cr_r, Cp_r\}\}$. Here, the set's elements represent the output and input channel port statuses of type `integer`. In the transition rules, for brevity we use the array indexing notation to refer to the channel and signal statuses: $E_c[w_s]$ represents the output channel port's write-sent status w_s (indicating that the channel *output port* is ready to send), while w_r (indicating that the channel *output port* is ready to receive an acknowledgment) and p_s (indicating that *output port* has been preempted) are the write-received and preemption statuses, respectively. Similarly for the input channel port, r_s, r_r, and p_r represent the read-sent (indicating that the *input port* is ready to receive), read-received (indicating that the *input port* is ready to send an acknowledgment), and preemption statuses, respectively. The receiving channel port statuses (r_r and p_r) are updated at the start of every local CD tick looking at the sending channel port statuses (w_s and p_s), while w_r is updated looking at the value of r_s. These statuses are used to carry out a full handshake when communicating via channels. Thus, the cardinality of set E_c is $3 \times 2 \times n$.

A statement p is said to be *selected* iff a **`pause`** is hit during the execution of p; a *selected* statement is decorated with a *hat*, e.g., $\hat{p}$. We use the notation $\bar{p}$ to indicate that the selected state for p is currently unknown. Also, a $\ddot{p}$ indicates that the position of the current control point over p is unknown. We refer the reader to Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.258, for the full definitions of $\hat{p}$, $\bar{p}$, and $\ddot{p}$.

The example below shows the microstep semantics of **`pause`** execution. The $\bullet$ represents the control point movement in the SystemJ program code. When a **`pause`** is hit for the first time (*also called the start rule*) the statement gets selected and the program ends with a termination code of 1. In the next instant (*also called the resumption rule*) the selected statement continues further and completes execution (termination code 0). The *selection* status is *upward-propagative*. Thus, any statement enclosing a **`pause`** is considered selected if the enclosed **`pause`** itself is currently selected. In the rules below, data stores have been omitted since we are dealing with a pure control statement

$$\bullet\, pause \xrightarrow[E]{1,\perp} \widehat{pause} \qquad \bullet\, \widehat{pause} \xrightarrow[E]{0,\perp} pause. \tag{2}$$

There are a number of other rewrite rules associated with reactive constructs of SystemJ; see Appendices B and C, available in the online supplemental material.

The macrostep transition rule for a SystemJ program is expressed in terms of the macrostep transition of individual CDs, which in turn is formed by combining the microstep rules for all the SystemJ constructs. The macrostep transition for a SystemJ program is

$$\bullet\, \bar{s_1} \underset{E_{s_1},E_{cs_1}}{\overset{e_{s_1},k_{s_1}}{\hookrightarrow}} \ddot{\bar{s}}_1, \quad \bullet \bar{s_2} \underset{E_{s_2},E_{cs_2}}{\overset{e_{s_2},k_{s_2}}{\hookrightarrow}} \ddot{\bar{s}}_2 \quad \ldots \quad \bullet\, \bar{s_m} \underset{E_{s_m},E_{cs_m}}{\overset{e_{s_m},k_{s_m}}{\hookrightarrow}} \ddot{\bar{s}}_m, \tag{3}$$

where, s_m is some CD and E_{s_m}, E_{cs_m}, e_{s_m}, and k_{s_m} are the signal sensitivity set, channel status sensitivity set, output signal set, and termination code for CD s_m.

4.2 Semantics of DSystemJ

Before describing the microstep rewrite rules for all DSystemJ syntactic constructs, we first present the equivalence between the DSystemJ MoC and SystemJ MoC, i.e., we define the behavior of a dynamic DSystemJ program in terms of a static SystemJ program.

4.2.1 DSystemJ Formal MoC

A DSystemJ program is equivalent to a SystemJ program if it has the same number of executing CDs, ($m = n$) and for every CD d_m in the DSystemJ program there exists a CD s_n in the SystemJ program, which when given an input signal set results in an equivalent macrostep transition and produces the same output signal set.

We define equivalence over a tick only. This is because a DSystemJ program may diverge in its behavior over an execution trace due to its ability to fork CDs at runtime.

4.2.2 Rewrite Rules for DSystemJ Syntactic Constructs

We now describe one by one the rewrite rules of the DSystemJ constructs presented in Table 2.

The $\rightarrow$ construct. Completes instantaneously with an exit code of 0 like any other instantaneous statement in SystemJ

$$\bullet\, \texttt{unique-name} \rightarrow \{\} \xrightarrow[E,E_c]{0,\perp} \texttt{unique-name} \rightarrow \{\} \tag{4a}$$

$$\bullet\, \texttt{unique-name} \rightarrow \texttt{cd} \xrightarrow[E,E_c]{0,\perp} \texttt{unique-name} \rightarrow \texttt{cd} \tag{4b}$$

The `run` construct. Does not have a single microstep rule. Instead, every **`run`** statement is rewritten into **`send`** and **`receive`** statements to perform a rendezvous with the runtime CD.

Consider an executor CD p running concurrently and asynchronously with the CD q, where p is:

```
receive C; m
```

`m` being the program code of some other CD `CD` and `C` being a unique named point-to-point channel between p and q, respectively. As a result, a program q:

```
run CD(args); emit S
```

can be rewritten as:

```
send C(args); emit S;
```

Both programs p and q take a transition τ, which is the macrostep rendezvous transition on channel C and the state change results in p transforming into `m`, while q transforms into **`emit`** `S`. The result is a system where the CD `CD` (`m`) runs in asynchronous parallel with the forking CD q after an extra transition τ. This is the required behavior of the **`run`** statement.

Informally, the semantics of the **`run`** statement assumes that every possible CD in the DSystemJ program is running but blocked on a **`receive`** `channel-name` statement, waiting for a successful rendezvous on the unique name

"channel-name" before proceeding further with its code. The **run** statement in turn performs a rendezvous with one of these CDs. Note that every **run** statement requires a fresh channel name.

The send and receive constructs. Implement CSP [5] style message passing. The difference with SystemJ is that we introduce the nondeterministic choice operator □, which chooses a rendezvous partner in case of a nonlinear rendezvous with multiple participants (see Section 2.2.3). This is similar to the **select** statement in ADA [17]

$$\{E_{c_p}[Cp_s] = E_{c_q}[Cp_r], E_{c_q}[Cr_r] > E_{c_q}[Cr_s], E_{c_r}[Cp_s] = E_{c_q}[Cp_r]\}/\{\{\bullet\hat{p}, data\square \bullet \hat{r}, data \overset{0,\perp}{\underset{E,E_c}{\hookrightarrow}} p, data' \bullet \hat{r}, data\}, \{\bullet\hat{q}, data \overset{0,\perp}{\underset{E,E_c}{\hookrightarrow}} q, data'\}\} \tag{5a}$$

$$\{E_{c_p}[Cp_s] = E_{c_q}[Cp_r], E_{c_q}[Cr_r] > E_{c_q}[Cr_s], E_{c_r}[Cp_s] = E_{c_q}[Cp_r]\}/\{\{\bullet\hat{p}, data\square \bullet \hat{r}, data \overset{0,\perp}{\underset{E,E_c}{\hookrightarrow}} r, data' \bullet \hat{p}, data\}, \{\bullet\hat{q}, data \overset{0,\perp}{\underset{E,E_c}{\hookrightarrow}} q, data'\}\}. \tag{5b}$$

Rules (5a) and (5b) show the macrostep rendezvous transition, when the rendezvous conditions are fulfilled, for two senders and a single receiver. The rules are read as follows: provided that no CDs are preempted, that $E_{c_p}[Cp_s]$ is equivalent to $E_{c_q}[Cp_r]$, and $E_{c_r}[Cp_s]$, and the receiving CD q is ready to rendezvous, (shown by $E_{c_q}[Cr_r] > E_{c_q}[Cr_s]$, where $>$ is the greater than operator), then the rendezvous takes place by making a nondeterministic choice between either of the sending CDs. The □ operator internally and nondeterministically chooses one of the sending CDs p (Rule (5a)) or r (Rule (5b)) to rendezvous with the receiving CD q. The other CD blocks waiting for an acknowledgment from the receiver. The r_r port status is updated at the start of every tick by looking at the w_s statuses of the sending CDs.

The rendezvous transition Rules (5a)-(5b) are valid only in the absence of strong preemptions, possible due to constructs such as an **abort**. DSystemJ's strong preemption in presence of rendezvous is similar to that of SystemJ's, except that a preemption can occur even before choosing a partner in case of multiparticipant rendezvous. Appendix D, available in the online supplemental material, gives the rest of the rendezvous rules, which deal with preemption.

The channel declaration construct. Has the same semantical rewrite rules as the SystemJ channel declaration statement. The differences between the two are purely syntactic: in DSystemJ, the **input** and **output** keywords, which define the input and output ports of the channel, are optional; the DSystemJ compiler infers the type of ports implicitly. Also, unlike SystemJ, DSystemJ allows new channel declarations at runtime.

4.3 Reactivity

Informally, a DSystemJ CD is reactive if, for every given input signal and channel sensitivity sets, there is at least a single macrostep transition that results in the production of an output signal set. Note that this output set might be empty. Formally, given a DSystemJ CD d_m, an input signal sensitivity set E_{d_m}, and a channel sensitivity set E_{cd_m}, d_m always takes a transition $\hookrightarrow^{e_{d_m},k_{d_m}}_{E_{d_m},E_{cd_m}}$, where $k_{d_m} \in \{0,1\}$. The definition for the reactivity of a SystemJ CD is identical.

Theorem. *Every DSystemJ CD is reactive.*

Proof sketch. The proof for reactivity of a SystemJ CD is based on the structural induction on the microstep rules, and requires proving that every microstep transition $\rightarrow$, contained in the macrostep $\hookrightarrow$, finishes with a termination code of $k \in \{0,1\}$. The simple but lengthy proof that a SystemJ CD is reactive is given in [19]. Then, as the DSystemJ MoC is defined in terms of SystemJ, all DSystemJ kernel constructs are rewritten into SystemJ constructs (Section 4.2.2). Thus, by implication, every DSystemJ CD is reactive. □

Rendezvous semantics (Rules (5a)-(5b) and Appendix D, available in the online supplemental material) of DSystemJ in conjunction with those of SystemJ are essential in guaranteeing reactivity in the presence of multiparticipant nonlinear rendezvous. As shown in the rendezvous rules, **send** and **receive** constructs when blocked, still take a microstep transition (Rule (31a)), producing a termination code k_{d_m} of 1. Intuitively, in the general case of an automaton, this is equivalent to having an implicit self-transition in each state. In the particular case of DSystemJ, this is equivalent to the reactive **await** statement waiting for an input signal. Indeed, the rendezvous in DSystemJ and SystemJ is implemented using the **await** statements working on the channel sensitivity set E_{cd_m}.

DSystemJ's syntactic constructs are rewritten into SystemJ constructs and hence, we follow the same compilation mechanism as in SystemJ. The microstep transitions presented in Section 4 lead to a direct intermediate representation, called the *Asynchronous GRaph Code* (AGRC). AGRC is an asynchronous extension of the *GRaph Code* (GRC) format used to compile Esterel [20]. The AGRC intermediate representation and the compilation of DSystemJ programs is presented in the next section.

5 Compiling DSystemJ Programs

Each DSystemJ CD is compiled into a single threaded Java program by the DSystemJ compiler. The formal semantics of encoding the DSystemJ programs into Java code is presented in Appendix E, available in the online supplemental material. These Java programs are then compiled with any Java compiler to create the class files. Next, the designer describes the underlying execution system topology and configuration via one or more XML files. This description of the system topology consists of information such as the IP addresses of the machines involved, binding of all the input/output signals to the underlying socket connections, etc. Finally, the DSystemJ program is launched by parsing this XML description into the DSystemJ runtime system. The runtime system needs to be started beforehand on all the machines that will participate in the system.

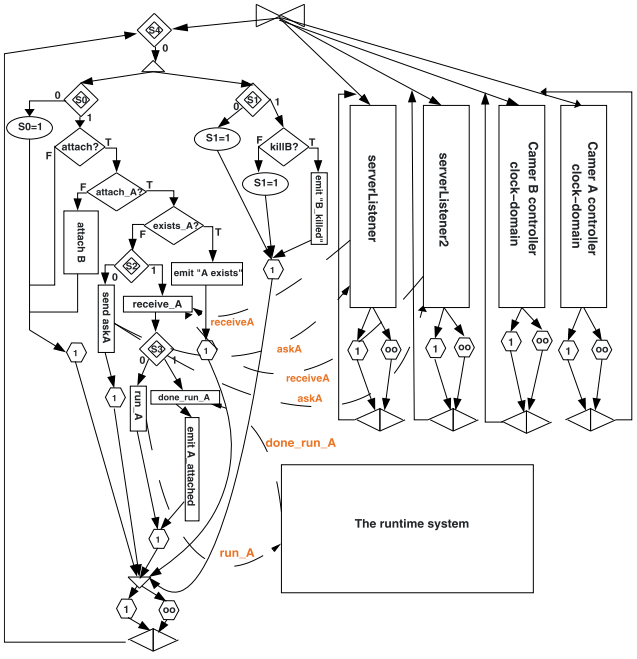
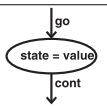
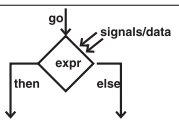
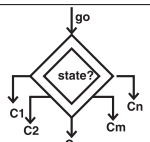
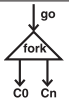
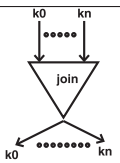
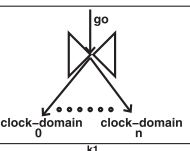
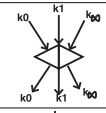


Fig. 8. The AGRC, for example, in listings 2 and 3.

TABLE 3
Description of Primitive Nodes

Primitive Node	Description
	Instantaneously carries out the "action" and sets the "cont" output to high when "go" is high.
	Encodes the state variable and instantaneously pass control to "cont".
	The test node checks for signal and data expressions. It instantaneously selects the "then" or "else" branch depending upon the evaluation of the expression.
	The state node instantaneously selects a branch $C_1 \ldots C_n$ depending upon the evaluation of the state variable.
	The fork node has a single input port "go" and a number of output ports, each connected to the input ports of the children synchronous parallel reactions. When triggered, the fork node instantaneously triggers all its output ports, thereby *forking* all the synchronous parallel reactions at once.
	The join node acts as the sync for the synchronous parallel reactions forked by the fork node. The input ports of the join node indicate all possible termination codes (k in micro-step transition semantics) of all synchronous parallel reactions that sync into this node. The join node invokes the continuation context by calculating the maximum termination code out of all the incident branches.
	The asynch-fork node instantaneously passes the control onto its children branches, thereby instantaneously forking the CDs.
	The asynch-join node acts as the sync for CDs. Unlike the join-node, the asynch-join node does not calculate the maximum termination code, and hence does not implement barrier synchronization like the join-node.
	The terminate node has one input and one output arc, "go" and "cont", respectively. Every statement in DSystemJ program completes with a termination code $k \in [0 \ldots \infty]$

5.1 The *Asynchronous GRaph Code*

DSystemJ compilation follows a structural translation scheme based on the *Structural Operational Semantics* (Section 4). Each DSystemJ statement is compiled into an intermediate representation. These representations combined together form a semantic preserving intermediate graph called the AGRC.

We show the intermediate translations of the kernel statements, necessary to understand the AGRC of Fig. 8. The AGRC is built a top the GRC representation used to compile Esterel [20]. Since AGRC includes the GRC format, the description of the AGRC incorporates the description of the GRC format.

Every node in the AGRC has named input and output ports. A node gathers through its input ports the control information needed for its computation: the *control input ports* receive the incoming control flow, the bullet (•) in our microstep transitions. The *signal input ports*, used only in the test nodes, collect the statuses of the signals involved in the test expression. The output ports are connected to input ports through control arcs. Each control arc connects one output port to one input port. When activated, the output port activates all incident control arcs. The class of control arcs that connect a signal emission to a signal test node is called a *signal dependency*. The signal dependency arcs ensure that a signal test is performed only after the emission of that signal. In case of read-write concurrency, we also have *data dependency* arcs: the data dependency arcs connect a variable write to a variable read, thereby ensuring data integrity. The interface of primitive nodes is shown in Table 3.

5.1.1 Translating the `pause` Statement

Every statement can have two possible behaviors; *Surface Flow Graph* (SFG) representing the first invocation of the program, and *Depth Flow Graph* (DFG) representing all other invocations (required due to schizophrenic signal behavior, see [15]). The `pause` statement's SFG finishes with a termination code of 1 according to the semantics as described in Rule (2). The DFG on the other hand finishes with a termination code of 0.

5.1.2 Translating the `emit` Statement

The **`emit`** statement sets the signal status it is emitting to high and finishes instantaneously with a termination code of 0.

5.1.3 Translating the `present` Statement

The **`present`** statement checks the expression `expr` and then takes the appropriate branch. The **`present`** statement is similar to **`if/else`** statements common in general purpose programming languages. Other signal-based statements such as **`await`**, **`abort`**, and **`suspend`** are all translated into test nodes like the one in the **`present`** statement and hence, have similar translations. Note that the tokens `m` and `n` represent the termination codes for the different branches, respectively.

5.1.4 Translating the || Operator

The synchronous parallel operator (||) forks two or more synchronous parallel reactions with the fork node. These reactions proceed in lockstep. If any of the forked reaction pauses, the DFG of the || operator is activated in the next logical tick. In the DFG, the switch node (double diamond) chooses the appropriate child branch to execute, depending upon its value, which is set by the enter node (ellipse) in the SFG. All the forked reactions complete with a termination code. These termination codes are sinked into a join node, which calculates their max, and this branch is taken as the continuation context. The join-node, along with the synchronizer (trapezoid) guarantees the lockstep execution of the synchronous parallel reactions.

5.1.5 Translating the >< Operator

The asynchronous parallel operator (><), which forks CDs, has similar translation to the || operator, except; there is no synchronization, i.e., once the topmost CD finishes with a termination code, it is rerun with the next set of input signals from the environment.

5.2 AGRC Representation of the Security Surveillance System

The translations described above can be combined together to form the AGRC, as shown in Fig. 8 for Listings 2 and 3. We now show how this AGRC captures the semantics of the DSystemJ program by traversing through the AGRC as control point movements. Fig. 8 is an abstract representation of Listings 2 and 3. The `GUIListener` CD is represented in some detail, while the other CDs from Listing 2 are completely abstracted out. Every CD starts with a switch node. The `GUIListener` CD decodes the switch node `S4` with a value 0, thereby entering its only child branch. Next, the fork node forks out two synchronous parallel reactions. The scheduling order of these two synchronous parallel reactions is inconsequential. Supposing that the left synchronous parallel reaction gets scheduled first, the `S0` switch node in the first synchronous parallel reaction again gets decoded to 0 and enters its left child. The enter node encodes the `S0` value as 1 and finishes execution with a exit code of 1 (indicating the completion of a tick). Similarly, the second synchronous parallel reaction completes a tick after encoding `S1` to 1. The join-node makes sure that all the incoming reactions complete with some exit code, thereby implementing the lockstep execution of the synchronous parallel reactions. Finally, the control reaches the asynch-join node, where the output signals, if any, are emitted to the environment and the new input signals are read from the environment. This represents the end of the tick transition for each CD. After reading the new set of input signals, a new iteration of the CD in a new tick is carried out.

In the second iteration, the first synchronous parallel reaction's `S0` switch node enters its right most child, where it first checks if the `attach` signal is present. If present, a check on the value of this signal is made. If the `attach` signal asks the `GUIListener` to attach the controller for camera A, the `GUIListener` then checks if this controller is already attached or not. If so, a signal with the string "`A exists`" is emitted, otherwise, this CD carries out a rendezvous with the either of the `serverListener` or `serverListener2` CDs via channels `askA` and `receiveA`, obtaining the code for the controller. Once the code is received, this CD makes a rendezvous with the runtime system via channels `run_A` and `done_run_A`, asking it to fork this recently obtained CD.

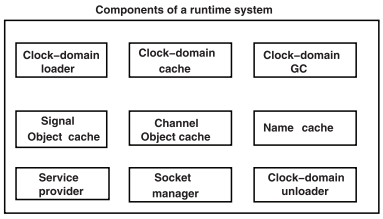


Fig. 9. The various CDs implementing the runtime system.

This is the expected behavior of the `GUIListener` CD shown in Listing 3. Listing 3 first waits to obtain an `attach` signal (line 7). Once obtained, the value of this signal is checked to see which camera controller needs to be forked (lines 10, 21). If the controller for camera A needs to be forked, an enquiry is made with the runtime system to see if this controller is already present (line 12). If this check succeeds, a message "`A exists`" is emitted to the environment, otherwise, a rendezvous is carried out with either the `serverListener` or `serverListener2` CD (lines 16-18), and finally the obtained camera controller code is instantiated as a CD (line 19).

In Fig. 8, the dotted lines show the channel communication. Most of the DSystemJ syntactic constructs from Table 2 are directly converted into primitive nodes of the AGRC. Only the **`send`**, **`receive`**, and **`run`** statements do not have a direct translation to primitive AGRC nodes. The **`send`** and **`receive`** statements are rewritten into algorithms operating on channel statuses using other reactive constructs [15] (namely **`await`** and **`emit`**) to implement a rendezvous. The **`run`** statement is rewritten as a combination of **`send`** and **`receive`** constructs making a rendezvous with the runtime system (Section 6.1). The runtime system itself is a GALS program responsible for CD management (see next section). The rendezvous in DSystemJ takes finite time to finish but the bound on this time cannot be computed statically, and thus, a **`run`** statement completes, in the process forking a CD, after a finite number of ticks of the CD that calls the **`run`** statement.

The Java code produced from the above AGRC is presented in Appendix F, available in the online supplemental material.

6 Runtime System and Library

The DSystemJ language compiler is accompanied with a runtime system (Fig. 9) and a library (Fig. 10) that is responsible for managing CDs. In this section, we give an overview of these components, which form a part of the complete DSystemJ runtime hierarchy. Our runtime system is around 106 KB in size, which makes it adequately small for today's high-end embedded system such as gaming consoles and smart phones. DSystemJ library's small size also makes

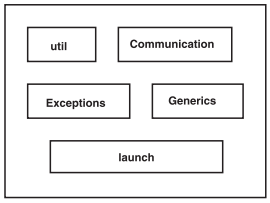


Fig. 10. The DSystemJ library support.

it capable of running on smaller embedded systems, especially, without support of operating systems [21].

6.1 The Runtime System

The CD loader loads the CD (compiled class files) when requested. It also registers the name of the CD and the related signal and channels with the name cache, and it loads the CD code along with the related signals and channels with the CD, signal object, and channel object caches, respectively. The CD unloader, unloads the CD from the CD cache and deregisters the CD and related signals and channels from the name cache, and finally calls the CD garbage collector (GC). The CD GC is essentially the Java GC and is used to free the allocated heap memory when not in use. The service provider is a utility class. DSystemJ programs can get access to the library functions via the service provider. Lastly, the socket manager manages the socket allocation and deallocations. Recall that sockets are the underlying communication mechanism for channels and interface signals.

6.2 The Library Support

The DSystemJ library provides a number of general and some specific classes (Fig. 10), which can be used by the designers to write DSystemJ programs more easily. The library is designed with the purpose of being easily extensible by designers. The util, Communication, Exceptions, and Generics classes provide utility functions, functions to communicate via TCP/IP and multicast, special extensible exceptions, and interfaces for extension of the runtime system, respectively. Launch is a special class: it launches the DSystemJ program on different machines. The launch library uses an XML-based description of the underlying execution architecture, which includes:

1. the description of the machines involved,
2. their IP addresses,
3. the ports used for communication,
4. the type of channels to be used,
5. the serializeR required to marshal data when communicating via channels, etc.

The launch library parses the XML file, looks up the required CD, and instantiates it for execution. For a complete description of the launch library, the reader is referred to Appendix G, available in the online supplemental material.

6.3 The Rendezvous Protocol

DSystemJ uses rendezvous via channels to enable communication between reactions in different CDs. A channel consists of two ports, the sending port and the receiving port. Each port is endowed with channel statuses, which are used to implement a handshake, described formally in Section 4.2. The channel statuses are sampled at the start of the tick and emitted to other CDs at the end of the tick. Thus, channel statuses can be considered equivalent to signals in DSystemJ. Hence, our rendezvous implementation uses a number of reactive kernel constructs like **`await`** and **`emit`** on the channel statuses [15].

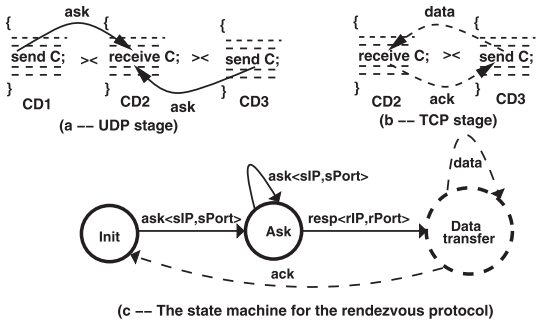


Fig. 11. The rendezvous protocol in DSystemJ.

Unlike SystemJ, DSystemJ allows multiparticipant rendezvous, i.e., a rendezvous with multiple senders or receivers. Fig. 11 shows an example rendezvous between two senders (CD1 and CD3) and a single receiver (CD2). We use a combination of UDP and TCP/IP to implement rendezvous in multiparticipant scenarios. Usually, the designer specifies in the XML description of the DSystemJ program the type of rendezvous that needs to be implemented (e.g., Appendix G, available in the online supplemental material, Listing 5, lines 8, 15, 23, 27, and 32). A UDP-based rendezvous should be chosen by the designer if a multiparticipant scenario is expected. Indeed, it is impossible to carry out a multiparticipant rendezvous without the UDP support; in contrast, TCP/IP-based rendezvous is more appropriate in the case of point-to-point rendezvous, because it is faster and more scalable, since the packets are not broadcasted to a whole subnet like in UDP-based communication.

Since the UDP-based rendezvous encompasses the TCP/IP-based rendezvous, we provide a UDP example. In Fig. 11a there are two senders trying to synchronize on the channel named C with a single receiver. Both senders broadcast the request to rendezvous (`ask` signal) on channel C. The receiver listening to this broadcast sends a reply back to a single sender, chosen *nondeterministically*. This nondeterministic choice (□ operator, Section 4.2.2) is implemented on a first come first serve basis. Fig. 11b shows the receiver choosing the second sender (CD3). Once a pairing is established, the sender and receiver carry out a two-phase handshake to synchronize and deliver data, if any (Fig. 11b).

Fig. 11c shows the state machine for the complete rendezvous protocol. The solid arrows and circles show the UDP communication, while the dashed arrows and circles show the TCP/IP communication.

Let us now consider the rendezvous scenario in Fig. 11c in more detail. In the `Init` state, the receiver listens on the multicast IP address specified in the XML description. The senders, on the other hand, broadcast using the multicast clients, while at the same time also listening on their TCP/IP ports (`sPort`) and IP address (`sIP`) allocated by the socket manager. The senders broadcast a request to

TABLE 4
Qualitative Comparison between DSystemJ and Other Similar Languages and Libraries

Languages and Libraries	Process Forking	SR-constructs	Hierarchical concurrency	Asynch-constructs	Mobility	Formal-MoC	Heterogeneity	Starvation avoidance
MPI	No	No	No	Yes	No	partial	Yes	No
RML	Yes	Yes	No	No	No	Yes	Yes	Yes
JoCaml	Yes	No	No	Yes	No	Yes	Yes	Yes
Occam-π	Yes	No	No	Yes	strong	Yes	No	No
Erlang / Salsa / Scala	Yes	No	No	Yes	partial weak mobility support	No	Yes	Yes
Actor Foundry	No	No	No	Yes	weak	Yes	Yes	Yes
JADE	Yes	No	No	Yes	strong	No	Yes	No
Concurrent-ML	Yes	No	No	Yes	No	Yes	Yes	Yes
Concurrent-Haskell	Yes	No	No	Yes	No	Yes	Yes	No
DSystemJ	**Yes**	**Yes**	**Yes**	**Yes**	**weak**	**Yes**	**Yes**	**No**

rendezvous (`ask`) along with `sIP` and `sPort` at the end of every tick.

At the start of its tick the receiver samples incoming data on the multicast server. If the receiver receives more than one rendezvous request (`ask`), the receiver chooses a sender *nondeterministically* (on a first come first serve basis). The receiver then stops listening on the multicast server and instead initializes a new TCP/IP server to start listening on. The TCP/IP-address (`rIP`) and the port number (`rPort`) are allocated by the socket manager. The receiver sends a TCP/IP reply to the sender using the `sIP` and `sPort` as the destination address and port. This reply consists of the `rIP` and `rPort` parameters.

Each sender (CD1 and CD3) samples incoming packets on the TCP/IP server at the start of its tick. Once the sender receives the receiver's `rIP` and `rPort` identifiers, everything needed to carry out a rendezvous is now available. Next, the sender and receiver carry out a two-phase handshake using the algorithms described in [15].

After the completion or preemption of this rendezvous, the receiver stops listening on its TCP/IP server (in the process freeing and recycling the sockets) and starts listening on the multicast UDP server again, ready to carry out another rendezvous when requested. This socket tear down and rebuilding takes place only in the case of multiparticipant rendezvous, in case of point-to-point rendezvous the server/clients are started once at the start of the CD and are alive until the CD dies.

7 Comparison with Other Languages and Libraries

Table 4 compares the qualitative properties between the different languages and libraries found in literature. **Process Forking**, is the ability to provide mechanisms to easily express the dynamic process creation. **SR-constructs** is ability of the language to incorporate data fusion capabilities as first class citizens of the language. **Hierarchical concurrency** is the ability of a language to allow designers to program multiple hierarchical concurrent levels with ease. **Asynch-constructs** defines the ability to program asynchronous distributed and multicore platforms. **Mobility** is the ability to describe movement of program code and possibly data on geographically distinct machines. **Formal-MoC**, is the property that the language is based on rigorous mathematical foundations. **Heterogeneity** is the property that control and data dominated applications can both be described and combined with ease. Finally, **Starvation avoidance** describes the guarantee that there is no starvation in the system.

MPI [2] is the de facto industry standard for programming distributed systems, but being a library rather than a language it lacks abstraction and does not provide a rigorous formal MoC. There have been recent attempts to provide a formal semantics for MPI [22], [23], [24], but these do not cover the MPI specification comprehensively.

RML [25] and JoCaml [10], both formal programming languages, are based on very different concepts. RML provides abstraction and SR data fusion constructs like DSystemJ, but lacks support for asynchronous processes and mobility. JoCaml is based on join-calculus [9] and is targeted at design and implementation of distributed programs, but lacks support for reactivity and mobility.

Occam-π [8] is based on the π-calculus [26] and hence is closest to DSystemJ, but, unlike DSystemJ, it does not support implementation of heterogeneous designs (with significant data dominated computations); also, it does not provide any reactive constructs as first class citizens of the language.

Erlang [14], Salsa [27], Scala [13], ActorFoundry [12], and JADE [3] are all based on the actor model of computation [11]. Erlang, Scala, and Salsa do not support process mobility as first class citizens of the language, while ActorFoundry and JADE support weak and strong mobility as language programming paradigms, Salsa and Scala pass references rather than copies of program code or of messages, which can contain program code and hence, are unable to accomplish mobile distributed processing. Concurrent-ML (CML [28]) provides CSP style communication mechanism along with dynamic process forking to the ML language. CML is similar to DSystemJ in its treatment of asynchronous communication constructs. DSystemJ and CML both provide CSP style rendezvous as a basic communication primitive, they both provide process

TABLE 5
Examples, *Lines of Code*, Generated Memory Footprint, and Total Memory Footprints

Examples	LOC		Generated memory foot-print (KB)		Total memory foot-print, including libraries (KB)	
	DSystemJ	JADE	DSystemJ	JADE	DSystemJ	JADE
send-receive	**39**	118	38	**5.6**	**145**	2616.6
camControl	**1594**	1927	753	**558.5**	**920**	3214
sieve	**163**	267	99	**12**	**216**	2623

mobility, since CML is a higher order language, and finally they both provide powerful data manipulation constructs for programming heterogeneous systems. DSystemJ's hierarchical concurrency mechanism (encapsulation of synchronous parallel reactions using the ‖ operator within asynchronous CDs) makes DSystemJ a more elegant language compared to CML. For example, CML requires the designers to use nonblocking **`transmit`** and **`receive`** in combination with **`sync`** on events produced by these constructs to implement blocking **`send`**/**`receive`** communication on channels. This design choice stems from the fact that every process in CML is single threaded, so, an indefinitely blocking **`receive`** might block all other constructs from proceeding further and hence block the whole process. This difficulty is overcome by introducing nonblocking **`transmit`**, **`receive`** along with **`sync`**, **`choose`**, **`wrap`**, and **`guard`** primitives in the language. DSystemJ, on the other hand, never faces these issues: potentially blocking receives and sends can be assigned to different synchronous parallel reactions. This, along with the guarantee of reactivity (see Section 4.3), provides a more abstract environment for programming concurrent systems. Concurrent-Haskell [29] is yet another language that provides concurrency to the Haskell programming language. Concurrent Haskell uses a concept called `M-var` to provide concurrent communication mechanisms between Haskell processes. `M-vars` are similar to semaphores in the Java language. Concurrent-Haskell like DSystemJ allows dynamic process forking and implementing rendezvous based on channels and other types of communication mechanisms using the `M-var` communication primitive. But this, in our opinion, makes the language less abstract compared to DSystemJ. In fact, `M-vars` being similar to Java's synchronized construct, allow the same level of abstraction (or lack thereof) and hence, make programming concurrent systems harder. Besides, none of these approaches provide the SR-programming paradigm. Finally, we would like to mention that, although a designer can introduce starvation (Section 2.2.3) in a DSystemJ program, this can be very easily avoided by using separate channels for communication between different pairs of processes.

8 Experimentation Results

In this section, we quantitatively compare DSystemJ with JADE. We chose JADE for comparisons, because the released version of ActorFoundry [12] does not support distributed implementations, and MPI-based Java bindings do not support process forking and mobility. We did not compare with C/C++ based implementations targeting distributed computing, because of course, such implementation would be faster than DSystemJ but, these systems lack the important quality of portability (they need to be recompiled every time with a different set of libraries for different underlying architectures and operating systems), which is required in our experimental setup. We also compared DSystemJ and SystemJ implementations (microbenchmarking) for concepts such as mobility, dynamic process creation, etc. These microbenchmarks provide an insight into the trade-offs between designer productivity and execution times. All the benchmarks, DSystemJ compiler, and runtime library are available to download from [30].

First we present the comparison results between DSystemJ and JADE. Table 5 shows the examples that we have chosen for comparison. We chose three very different programs; 1) `send-receive` is a simple communicator, that sends and receives continuously between two CDs in a very tight loop. Both the CDs in `send-receive` are static, i.e., forked at the start of the program. This program judges the communication performance. 2) `Sieve`, is the classical sieve of Eratosthenes, which computes primes. The `Sieve` example involves a large amount of process forking: except for the parent CD, which forks all the other CDs, CDs are forked multiple times dynamically, and communication between the various CDs is also established dynamically. Finally, 3) `camControl` is the example shown previously in Listings 2 and 3, expanded to 100 cameras; it involves a significant amount of code mobility across machines in a network, in conjunction with dynamic process forking. The experimental setup consists of: 1) a two-core 32-bit Linux machine running Sun-jdk-1.6 and 2) a two-core 64-bit Linux machine running open-jdk-1.6. All the Java class files were compiled using Sun javac-1.6 compiler.

As can be seen from Table 5, DSystemJ performs well compared to JADE. DSystemJ's abstract syntactic constructs along with its formal MoC help the designer to write code succinctly. JADE, being a library, lacks these advantages and, hence, requires more lines of code. DSystemJ also performs better than JADE with regards to the total memory footprint (class files). This advantage can be attributed to the tiny DSystemJ library footprint (106 KB for DSystemJ compared to 2.6 MB for JADE) as opposed to the generated code size. DSystemJ compiler produces bigger Java files and consequentially larger class files, unlike the hand written Java files as is the case with JADE.

Tables 6 and 7 show the runtime comparison between DSystemJ and JADE. Table 6 shows the runtime for a single 32-bit machine implementation with two cores. We ran the `send-receive` and `sieve` examples on this platform for a million ticks to get the results. The runtime is in `ms/tick`. DSystemJ has a clear notion of a tick; for JADE, all agents were implemented with `CyclicBehaviour` class (any excess runtime penalty associated with using this behavior in JADE is accounted for in Tables 6 and 7), which implements reactivity, to emulate the same behavior as in

TABLE 6
Runtime Comparison between DSystemJ and JADE on a Single Machine with Two Cores

Examples	Runtime (ms/tick)									
	DSystemJ					JADE				
send-receive	CD1	CD2				CD1	CD2			
	5	**5.57**				74.7	185.9			
sieve	CD1	CD2	CD3	CD4	CD5	CD1	CD2	CD3	CD4	CD5
	0.1	**17**	**16.75**	**23.4**	**17**	1	340	361.8	322.435	514

TABLE 7
Runtime Comparison between DSystemJ and JADE on a Distributed Platform with Two Machines and Four Cores

Examples	Runtime (ms/tick)									
	DSystemJ					JADE				
send-receive	CD1	CD2			CD1	CD2				
	20.7	**22.2**			86.88	470				
camControl	CD1	CD2	CD3	CD4	CD5	CD1	CD2	CD3	CD4	CD5
	342.7	**491.6**	**165.4**	**123.2**	**101.1**	3201.4	1376.8	1426.6	1578	1497.5

TABLE 8
Microbenchmarking: SystemJ versus DSystemJ

Comparison criteria	SystemJ		DSystemJ	
	LOC	ms/tick	LOC	ms/tick
Static CD forking (><)	1	23	1	23
Dynamic CD forking (**run**)	N/A	N/A	1	24
Mobility	22	50	2	70
Point-to-Point channel communication	2	Receiver CD : 0.1/Sender CD : 0.2	2	Receiver CD : 5/Sender CD : 5.2
One-to-Many channel communication	N/A	N/A	2	Receiver CD : 145/Sender CD : 152

DSystemJ. DSystemJ is far superior compared to JADE implementations. The slow JADE runtimes can be attributed to the fact that JADE implements a much more elaborate communication, dynamic forking, and mobility protocol compared to DSystemJ, based on the FIPA [31] standard. Also, while DSystemJ runtime library is optimized for single machine implementations, JADE concentrates more on the transparent locality model, i.e., the same communication mechanism is used for single and distributed platforms and this difference in implementation affects the runtime results.

In a distributed setting (Table 7), DSystemJ again outperforms JADE, although, in this case, the performance difference is smaller. This poor performance of JADE can again be attributed to the elaborate communication and mobility protocols that one has to follow when implementing JADE agents. On average, DSystemJ is 20 times faster compared to JADE on a single machine (multicore) implementation, and 12 times faster in a distributed setting.

Table 8 shows the microbenchmarks comparing DSystemJ and SystemJ. The comparison criteria consists of the new statements and concepts introduced in DSystemJ. The static CD forking in both SystemJ and DSystemJ is described using the >< operator, which behaves similarly in both and has the exact same runtime. DSystemJ allows designers to fork CDs dynamically using the **run** statement. It is impossible to do this in the SystemJ language. One can bypass this constraint in SystemJ by using Java, but this would break the semantics of the language itself and hence is not encouraged. As we can see from Table 8, dynamically forking a CD takes about the same time as static CD forking. The main drawback is that we don't know in which tick this CD will start running, because **run** is implemented as a rendezvous. Whereas in case of >< all CDs are forked from the start of the program. Both SystemJ and DSystemJ allow the mobility of CDs. In SystemJ a designer needs to handcode the mobility, i.e., wrapping the CD compiled class into a generic class, which can be loaded, marshaling the CD code, etc. DSystemJ provides built in constructs for mobility and thus significantly reduces the overall designer effort (as seen from the LOC column). The runtime for both hand marshaled code and that done via the DSystemJ runtime and library are comparable (2 ms/tick). SystemJ beats DSystemJ when comparing channel-based communication. In SystemJ, all communication takes place via shared memory, so, passing data via channels just amounts to passing a reference pointer, whereas in DSystemJ all communications take place via sockets, which involves marshaling the data being sent and copying the contents from one CD to the other. Finally, the one-to-many communication introduced in DSystemJ is much more expensive compared to point-to-point communication, because of the combination of UDP-based broadcast and TCP/IP, and especially establishing the TCP/IP server/client sockets as described in Section 6.3.

9 Conclusion and Future Work

In this paper, we have described a new programming language called DSystemJ, designed specifically for *dynamic distributed systems*. DSystemJ has rigorous mathematical semantics and hence, is amenable to compilation and formal reasoning. DSystemJ compared to other formal languages in its application domain provides an exhaustive design perspective, taking distributed communication, mobility, dynamic forking, and reactivity into account, thereby easing the design burden of software programmers. DSystemJ's operational semantics can be used to derive the behavioral semantics and thus, is helpful in abstract modeling and formal verification (out of the scope of this paper). The

formal semantics also allow us to build a compiler, which guarantees correct by construction implementation.

In the future, we plan to provide tools for formal verification and real-time analysis of DSystemJ programs. We also plan to use the presented operational semantics to derive distributed controllers for addressing the fairness and nondeterministic behavior of DSystemJ programs raised in this paper.

REFERENCES

[1] E. Lee, "The Problem with Threads," *Computer,* vol. 39, no. 5, pp. 33-42, May 2006.

[2] M.J. Quinn, *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill, Inc., 2004.

[3] "The JADE Website," http://jade.tilab.com, 2010.

[4] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," *Proc. IEEE Fifth Ann. Symp. Logic in Computer Science (LICS '90),* pp. 428-439, June 1990.

[5] C.A.R. Hoare, *Communicating Sequential Processes.* Prentice Hall, 1985.

[6] J. Galletly, *Occam-2,* second ed. Univ. College London Press, 1996.

[7] R. Milner, *Communication and Concurrency.* Prentice-Hall, Inc., 1989.

[8] P. Welch and F. Barnes, "Communicating Mobile Processes: Introducing Occam-pi," *Proc. Symp. Occasion of 25 Years of Comm. Sequential Processes (CSP),* A. Abdallah, C. Jones, and J. Sanders, eds., pp. 175-210, http://www.cs.kent.ac.uk/pubs/2005/2162, Apr. 2005.

[9] C. Fournet and G. Gonthier, "The Reflexive CHAM and the Join-Calculus," *Proc. 23rd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '96),* pp. 372-385, 1996.

[10] L. Mandel and L. Maranget, "Programming in JoCaml - Extended Version," Technical Report 6261, INRIA, 2008.

[11] W. Clinger, "Foundations of Actor Semantics," PhD dissertation, Massachusetts Inst. of Technology, 1981.

[12] "ActorFoundry," http://osl.cs.uiuc.edu/af/, 2010.

[13] M. Odersky, P. Altherr, V. Crement, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger, "An Overview of the Scala Programming Language," Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[14] R. Virding, C. Wikstrom, and M. Williams, *Concurrent Programming in Erlang,* second ed. Prentice Hall PTR, 1996.

[15] A. Malik, Z. Salcic, P.S. Roop, and A. Girault, "SystemJ: A GALS Language for System Level Design," *Elsevier J. Computer Languages, Systems and Structures,* vol. 36, no. 4, pp. 317-344, Dec. 2010.

[16] G. Berry, "The Semantics of Pure Esterel," citeseer.ist.psu.edu/berry93semantics.html, 1993.

[17] L. Henry, *Reference Manual for the ADA Programming Language.* Springer-Verlag, 1983.

[18] A. Malik, Z. Salcic, and P.S. Roop, "SystemJ Compilation Using the Tandem Virtual Machine Approach," *ACM Trans. Design Automation of Electronic Systems,* vol. 14, no. 3, pp. 1-37, 2009.

[19] A. Malik, "Principia Lingua SystemJ," PhD dissertation, Univ. of Auckland, 2010.

[20] D. Potop-Butucaru, S.A. Edwards, and G. Berry, *Compiling Esterel.* Springer Verlag, May 2007.

[21] A. Malik, Z. Salcic, A. Girault, A. Walker, and S.C. Lee, "A Customizable Multiprocessor for Globally Asynchronous Locally Synchronous Execution," *Proc. Seventh Int'l Workshop Java Technologies for Real-Time and Embedded Systems (JTRES '09),* pp. 120-129, 2009.

[22] S.F. Siegel and G.S. Avrunin, "Verification of MPI-Based Software for Scientific Computation," *Proc. 11th Int'l SPIN Workshop Model Checking Software,* S. Graf and L. Mounier, eds., pp. 286-303, 2004.

[23] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R.M. Kirby, and R. Thakur, "Formal Verification of Practical MPI Programs," *Proc. 14th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '09),* pp. 261-270, 2009.

[24] G. Li, M. Delisi, G. Gopalakrishnan, and R.M. Kirby, "Formal Specification of the MPI-2.0 Standard in TLA+," *Proc. 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '08),* pp. 283-284, 2008.

[25] L. Mandel and M. Pouzet, "ReactiveML: A Reactive Extension to ML," *Proc. Seventh ACM SIGPLAN Int'l Conf. Principles and Practice of Declarative Programming (PPDP),* pp. 82-93, 2005.

[26] R. Milner, J. Parrow, and D. Walker, "A Calculus of Mobile Processes, I," *Information and Computation,* vol. 100, no. 1, pp. 1-40, 1992.

[27] C. Varela and G. Agha, "Programming Dynamically Reconfigurable Open Systems with SALSA," *ACM SIGPLAN Notices,* vol. 36, no. 12, pp. 20-34, Dec. 2001.

[28] J.H. Reppy, "CML: A Higher Concurrent Language," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '91),* pp. 293-305, http://doi.acm.org/10.1145/113445.113470, 1991.

[29] S.P. Jones, A. Gordon, and S. Finne, "Concurrent Haskell," *Proc. Ann. Symp. Principles of Programming Languages (POPL '96),* pp. 295-308, 1996.

[30] "The DSystemJ Website," http://dsystemj.gforge.inria.fr, 2010.

[31] "The Foundation for Intelligent Physical Agents," http://www.fipa.org, 2010.

[32] K. Havelund and T. Pressburger, "Model Checking Java Programs Using Java PathFinder," *Int'l J. Software Tools for Technology Transfer,* vol. 2, no. 4, pp. 366-381, 2000.

Avinash Malik received the bachelor of engineering (BE) degree in electrical and electronic engineering and the PhD degree in electrical and computer systems engineering from the University of Auckland, NZ. He works in the field of programming language design and implementation. His main research interest lies in programming languages for multicore and distributed systems and their formal semantics/compilation. He has designed and implemented multiple languages based on the formal Globally Asynchronous Locally Synchronous model of computation. He has publications in a wide range of fields such as; programming language design and formal semantics, real-time systems, compilation, virtual machines, operating systems, and processor design. He has worked at organizations such as INRIA in France, Trinity College Dublin, IBM research Ireland, and IBM Watson on design and compilation of programming languages.

Alain Girault received the PhD degree while at the Verimag laboratory in 1994, and the habilitation in 2006 while at INRIA. He is a researcher at INRIA. He has held visiting positions in the ESTEREL team in France, in the PTOLEMY group at UC Berkeley, and at the University of Auckland. His research interests include the design of embedded and reactive systems, with a special concern for programming languages, distributed implementation, fault-tolerance, reliability, low-power, and multicriteria optimization. He heads the POP ART team at INRIA in Grenoble, which focuses on formal methods for embedded systems.

Zoran Salcic (SM'96) received the BE (1972), ME (1974), and PhD (1976) degrees in electrical engineering from the University of Sarajevo. He did most of the PhD research at the City College New York (CCNY). He is a professor of computer systems engineering with the Department of Electrical and Computer Engineering, University of Auckland, New Zealand. His main research interests include complex digital systems design, custom-computing machines, reconfigurable computing, FPGAs, processor and computer systems architectures, embedded systems and their implementation, design automation tools for embedded systems, hardware-software codesign, new computing architectures, models of computation and languages for concurrent and reactive systems, and related areas in computer systems engineering. He has published more than 240 peer-reviewed journal and conference papers, several books and numerous technical reports. He is an editor-in-chief of *EURASIP Journal on Embedded systems*. He is a fellow of the Royal Society (Academy of Science) New Zealand and recipient of Alexander von Humboldt Research Award in 2010. He is a senior member of the IEEE.